EPiC Series in Computing

Volume 40, 2016, Pages 52–66

IWIL-2015. 11th International Workshop on the Implementation of Logics

Well-founded Functions and Extreme Predicates in Dafny: A Tutorial

K. Rustan M. Leino

Microsoft Research
leino@microsoft.com

Abstract

A recursive function is well defined if its every recursive call corresponds a decrease in some well-founded order. Such *well-founded functions* are useful for example in computer programs when computing a value from some input. A boolean function can also be defined as an extreme solution to a recurrence relation, that is, as a least or greatest fixpoint of some functor. Such *extreme predicates* are useful for example in logic when encoding a set of inductive or coinductive inference rules. The verification-aware programming language Dafny supports both well-founded functions and extreme predicates. This tutorial describes the difference in general terms, and then describes novel syntactic support in Dafny for defining and proving lemmas with extreme predicates. Various examples and considerations are given. Although Dafny's verifier has at its core a first-order SMT solver, Dafny's logical encoding makes it possible to reason about fixpoints in an automated way.

0. Introduction

Recursive functions are a core part of computer science and mathematics. Roughly speaking, when the definition of such a function spells out a terminating computation from given arguments, we may refer to it as a *well-founded function*. For example, the common factorial and Fibonacci functions are well-founded functions. There are also other ways to define functions. An important case regards the definition of a boolean function as an extreme solution (that is, a least or greatest solution) to some equation. For computer scientists with interests in logic or programming languages, these *extreme predicates* are important because they describe the judgments that can be justified by a given set of inference rules (see, e.g., [2, 17, 20, 24, 27]).

To benefit from machine-assisted reasoning, it is necessary not just to understand extreme predicates but also to have techniques for proving theorems about them. A foundation for this reasoning was developed by Paulin-Mohring [22] and is the basis of the constructive logic supported by Coq [0] as well as other proof assistants [1, 25]. Essentially, the idea is to represent the knowledge that an extreme predicate holds by the proof term by which this knowledge was derived. For a predicate defined as the least solution, such proof terms are values of an inductive datatype (that is, finite proof trees), and for the greatest solution, a coinductive datatype (that is, possibly infinite proof trees). This means that one can use induction and coinduction when reasoning about these proof trees. Therefore, these extreme predicates are known as, respectively, *inductive predicates* and *coinductive predicates* (or, *co-predicates* for short). Support for extreme predicates is also available in the proof assistants Isabelle [23] and HOL [5].

In this paper, I give my own tutorial account on the distinction between well-founded functions and extreme predicates. I also show how the verification-aware programming language Dafny [12] sets these up to obtain automation from an underlying first-order (that is, fixpoint and induction ignorant) SMT solver. The encoding for coinductive predicates in Dafny was described previously [15]. The present paper adds inductive predicates (which are duals of the coinductive ones), new syntactic shorthands (based on the experience of using inductive predicates in Dafny), and examples.

1. Function Definitions

To define a function $f\colon X \to Y$ in terms of itself, one can write an equation like

$$f \quad = \quad \mathcal{F}(f) \tag{0}$$

where $\mathcal{F}$ is a non-recursive function of type $(X \to Y) \to X \to Y$. Because it takes a function as an argument, $\mathcal{F}$ is referred to as a *functor* (or *functional*, but not to be confused by the category-theory notion of a functor). Throughout, I will assume that $\mathcal{F}(f)$ by itself is well defined, for example that it does not divide by zero. I will also assume that f occurs only in fully applied calls in $\mathcal{F}(f)$; eta expansion can be applied to ensure this. If f is a boolean function, that is, if Y is the type of booleans, then I call f a *predicate*.

For example, the common Fibonacci function over the natural numbers can be defined by the equation

$$\mathit{fib} \quad = \quad \lambda n \bullet \text{ if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-2) + \mathit{fib}(n-1) \tag{1}$$

With the understanding that the argument n is universally quantified, we can write this equation equivalently as

$$\mathit{fib}(n) \quad = \quad \text{if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-2) + \mathit{fib}(n-1) \tag{2}$$

The fact that the function being defined occurs on both sides of the equation causes concern that we might not be defining the function properly, leading to a logical inconsistency. In general, there could be many solutions to an equation like (0) or there could be none. Let's consider two ways to make sure we're defining the function uniquely.

1.0. Well-founded Functions

A standard way to ensure that equation (0) has a unique solution in f is to make sure the recursion is well-founded, which roughly means that the recursion terminates. This is done by introducing any well-founded relation $\ll$ on the domain of f and making sure that the argument to each recursive call goes down in this ordering. More precisely, if we formulate (0) as

$$f(x) \quad = \quad \mathcal{F}'(f) \tag{3}$$

then we want to check $E \ll x$ for each call $f(E)$ in $\mathcal{F}'(f)$. When a function definition satisfies this *decrement condition*, then the function is said to be *well-founded*.

For example, to check the decrement condition for *fib* in (2), we can pick $\ll$ to be the arithmetic less-than relation on natural numbers and check the following, for any n:

$$2 \le n \quad \Longrightarrow \quad n-2 \ll n \quad \wedge \quad n-1 \ll n \tag{4}$$

Note that we are entitled to using the antecedent $2 \le n$, because that is the condition under which the else branch in (2) is evaluated.

A well-founded function is often thought of as "terminating" in the sense that the recursive *depth* in evaluating f on any given argument is finite. That is, there are no infinite descending chains of recursive calls. However, the evaluation of f on a given argument may fail to terminate, because its *width* may be infinite. For example, let P be some predicate defined on the ordinals and let $PDownward$ be a predicate on the ordinals defined by the following equation:

$$PDownward(o) \quad = \quad P(o) \ \wedge \ \forall p \bullet \ p \ll o \implies PDownward(p) \tag{5}$$

With $\ll$ as the usual ordering on ordinals, this equation satisfies the decrement condition, but evaluating $PDownward(\omega)$ would require evaluating $PDownward(n)$ for every natural number n. However, what we are concerned about here is to avoid mathematical inconsistencies, and that is indeed a consequence of the decrement condition.

1.0.0. Example with Well-founded Functions

So that we can later see how inductive proofs are done in Dafny, let's prove that for any n, $\mathit{fib}(n)$ is even iff n is a multiple of 3. We split our task into two cases. If $n < 2$, then the property follows directly from the definition of fib. Otherwise, note that exactly one of the three numbers $n-2$, $n-1$, and n is a multiple of 3. If n is the multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $\mathit{fib}(n-2) + \mathit{fib}(n-1)$ is the sum of two odd numbers, which is even. If $n-2$ or $n-1$ is a multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $\mathit{fib}(n-2) + \mathit{fib}(n-1)$ is the sum of an even number and an odd number, which is odd. In this proof, we invoked the induction hypothesis on $n-2$ and on $n-1$. This is allowed, because both are smaller than n, and hence the invocations go down in the well-founded ordering on natural numbers.

1.1. Extreme Solutions

We don't need to exclude the possibility of equation (0) having multiple solutions—instead, we can just be clear about which one of them we want. Let's explore this, after a smidgen of lattice theory.

For any complete lattice $(Y, \leq)$ and any set X, we can by *pointwise extension* define a complete lattice $(X \to Y, \ \dot{\Rightarrow} \ )$, where for any $f, g\colon X \to Y$,

$$f \ \dot{\Rightarrow} \ g \ \equiv \ \forall x \bullet \ f(x) \leq g(x) \tag{6}$$

In particular, if Y is the set of booleans ordered by implication ($\mathit{false} \leq \mathit{true}$), then the set of predicates over any domain X forms a complete lattice. Tarski's Theorem [26] tells us that any monotonic function over a complete lattice has a least and a greatest fixpoint. In particular, this means that $\mathcal{F}$ has a least fixpoint and a greatest fixpoint, provided $\mathcal{F}$ is monotonic.

Speaking about the *set of solutions* in f to (0) is the same as speaking about the *set of fixpoints* of functor $\mathcal{F}$. In particular, the least and greatest solutions to (0) are the same as the least and greatest fixpoints of $\mathcal{F}$. In casual speak, it happens that we say "fixpoint of (0)", or more grotesquely, "fixpoint of f" when we really mean "fixpoint of $\mathcal{F}$".

In conclusion of our little excursion into lattice theory, we have that, under the proviso of $\mathcal{F}$ being monotonic, the set of solutions in f to (0) is nonempty, and among these solutions, there is in the $\dot{\Rightarrow}$ ordering a least solution (that is, a function that returns false more often than any other) and a greatest solution (that is, a function that returns true more often than any other).

When discussing extreme solutions, I will now restrict my attention to boolean functions (that is, with Y being the type of booleans). Functor $\mathcal{F}$ is monotonic if the calls to f in $\mathcal{F}'(f)$ are in *positive positions* (that is, under an even number of negations). Indeed, from now on, I will restrict my attention to such monotonic functors $\mathcal{F}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{g(0)}}{g(2)}}{g(4)}}{g(6)} \qquad\qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\;\;}}{g(-5)}}{g(-3)}}{g(-1)}}{g(1)}$$

Figure 0. Left: a finite proof tree that uses the rules of (9) to establish $g(6)$. Right: an infinite proof tree that uses the rules of (10) to establish $g(1)$.

Let me introduce a running example. Consider the following equation, where x ranges over the integers:

$$g(x) \quad = \quad (x = 0 \;\vee\; g(x-2)) \tag{7}$$

This equation has four solutions in g. With w ranging over the integers, they are:

$$\begin{array}{lcl} g(x) & \equiv & x \in \{w \mid 0 \le w \;\wedge\; w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid (0 \le w \;\wedge\; w \text{ even}) \;\vee\; w \text{ odd}\} \\ g(x) & \equiv & x \in \{w \mid \mathit{true}\} \end{array} \tag{8}$$

The first of these is the least solution and the last is the greatest solution.

In the literature, the definition of an extreme predicate is often given as a set of *inference rules*. To designate the least solution, a single line separating the antecedent (on top) from conclusion (on bottom) is used:

$$\dfrac{}{g(0)} \qquad\qquad \dfrac{g(x-2)}{g(x)} \tag{9}$$

Through repeated applications of such rules, one can show that the predicate holds for a particular value. For example, the *derivation*, or *proof tree*, to the left in Figure 0 shows that $g(6)$ holds. (In this simple example, the derivation is a rather degenerate proof "tree".) The use of these inference rules gives rise to a least solution, because proof trees are accepted only if they are *finite*.

When inference rules are to designate the greatest solution, a double line is used:

$$\dfrac{}{\;g(0)\;} \qquad\qquad \dfrac{g(x-2)}{\;g(x)\;} \tag{10}$$

In this case, proof trees are allowed to be infinite. For example, the (partial depiction of the) infinite proof tree on the right in Figure 0 shows that $g(1)$ holds.

Note that derivations may not be unique. For example, in the case of the greatest solution for g, there are two proof trees that establish $g(0)$: one is the finite proof tree that uses the left-hand rule of (10) once, the other is the infinite proof tree that keeps on using the right-hand rule of (10).

1.2. Working with Extreme Predicates

In general, one cannot evaluate whether or not an extreme predicate holds for some input, because doing so may take an infinite number of steps. For example, following the recursive calls in the definition (7) to try to evaluate $g(7)$ would never terminate. However, there are useful ways to establish that an extreme predicate holds and there are ways to make use of one once it has been established.

For any $\mathcal{F}$ as in (0), I define two infinite series of well-founded functions, ${}^\flat f_k$ and ${}^\sharp f_k$ where k ranges over the natural numbers:

$$ {}^\flat f_k(x) \quad = \quad \begin{cases} \mathit{false} & \text{if } k = 0 \\ \mathcal{F}({}^\flat f_{k-1})(x) & \text{if } k > 0 \end{cases} \tag{11} $$

$$ {}^\sharp f_k(x) \quad = \quad \begin{cases} \mathit{true} & \text{if } k = 0 \\ \mathcal{F}({}^\sharp f_{k-1})(x) & \text{if } k > 0 \end{cases} \tag{12} $$

These functions are called the *iterates* of f, and I will also refer to them as the *prefix predicates* of f (or the *prefix predicate* of f, if we think of k as being a parameter). Alternatively, we can define ${}^\flat f_k$ and ${}^\sharp f_k$ without mentioning x: Let $\bot$ denote the function that always returns false, let $\top$ denote the function that always returns true, and let a superscript on $\mathcal{F}$ denote exponentiation (for example, $\mathcal{F}^0(f) = f$ and $\mathcal{F}^2(f) = \mathcal{F}(\mathcal{F}(f))$). Then, (11) and (12) can be stated equivalently as ${}^\flat f_k = \mathcal{F}^k(\bot)$ and ${}^\sharp f_k = \mathcal{F}^k(\top)$.

For any solution f to equation (0), we have, for any k and ℓ such that $k \leq \ell$:

$$ {}^\flat f_k \quad \dot{\Rightarrow} \quad {}^\flat f_\ell \quad \dot{\Rightarrow} \quad f \quad \dot{\Rightarrow} \quad {}^\sharp f_\ell \quad \dot{\Rightarrow} \quad {}^\sharp f_k \tag{13} $$

In other words, every ${}^\flat f_k$ is a *pre-fixpoint* of f and every ${}^\sharp f_k$ is a *post-fixpoint* of f. Next, I define two functions, $f^\downarrow$ and $f^\uparrow$, in terms of the prefix predicates:

$$ f^\downarrow(x) \quad = \quad \exists k \bullet {}^\flat f_k(x) \tag{14} $$

$$ f^\uparrow(x) \quad = \quad \forall k \bullet {}^\sharp f_k(x) \tag{15} $$

By (13), we also have that $f^\downarrow$ is a pre-fixpoint of $\mathcal{F}$ and $f^\uparrow$ is a post-fixpoint of $\mathcal{F}$. The marvelous thing is that, if $\mathcal{F}$ is *continuous*, then $f^\downarrow$ and $f^\uparrow$ are the least and greatest fixpoints of $\mathcal{F}$. These equations let us do proofs by induction when dealing with extreme predicates. I will explain in Section 2.2 how to check for continuity.

Let's consider two examples, both involving function g in (7). As it turns out, g's defining functor is continuous, and therefore I will write $g^\downarrow$ and $g^\uparrow$ to denote the least and greatest solutions for g in (7).

1.2.0. Example with Least Solution

The main technique for establishing that $g^\downarrow(x)$ holds for some x, that is, proving something of the form $Q \implies g^\downarrow(x)$, is to construct a proof tree like the one for $g(6)$ in Figure 0. For a proof in this direction, since we're just applying the defining equation, the fact that we're using a least solution for g never plays a role (as long as we limit ourselves to finite derivations).

The technique for going in the other direction, proving something *from* an established $g^\downarrow$ property, that is, showing something of the form $g^\downarrow(x) \implies R$, typically uses induction on the structure of the proof tree. When the antecedent of our proof obligation includes a predicate term $g^\downarrow(x)$, it is sound to imagine that we have been given a proof tree for $g^\downarrow(x)$. Such a proof tree would be a data structure—to be more precise, a term in an *inductive datatype*. For this reason, least solutions like $g^\downarrow$ have been given the name *inductive predicate*.

Let's prove $g^\downarrow(x) \implies 0 \leq x \ \wedge \ x$ even. We split our task into two cases, corresponding to which of the two proof rules in (9) was the last one applied to establish $g^\downarrow(x)$. If it was the left-hand rule, then $x = 0$, which makes it easy to establish the conclusion of our proof goal. If it was the right-hand rule, then we unfold the proof tree one level and obtain $g^\downarrow(x-2)$. Since the proof tree for $g^\downarrow(x-2)$ is smaller than where we started, we invoke the *induction hypothesis* and obtain $0 \leq (x-2) \ \wedge \ (x-2)$ even, from which it is easy to establish the conclusion of our proof goal.

Here's how we do the proof formally using (14). We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & f^{\uparrow}(x) \implies R \\
= & \quad \{ \text{ (14) } \} \\
 & (\exists k \bullet {}^{\flat}f_k(x)) \implies R \\
= & \quad \{ \text{ distribute } \implies \text{ over } \exists \text{ to the left } \} \\
 & \forall k \bullet ({}^{\flat}f_k(x) \implies R)
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove ${}^{\flat}g_k(x) \implies 0 \leq x \ \wedge \ x$ even for every k. If $k = 0$, then ${}^{\flat}g_k(x)$ is *false*, so our goal holds trivially. If $k > 0$, then ${}^{\flat}g_k(x) = (x = 0 \ \vee \ {}^{\flat}g_{k-1}(x-2))$. Our goal holds easily for the first disjunct ($x = 0$). For the other disjunct, we apply the induction hypothesis (on the smaller $k-1$ and with $x-2$) and obtain $0 \leq (x-2) \ \wedge \ (x-2)$ even, from which our proof goal follows.

1.2.1. Example with Greatest Solution

We can think of a given predicate $g^{\uparrow}(x)$ as being represented by a proof tree—in this case a term in a *coinductive datatype*, since the proof may be infinite. For this reason, greatest solutions like $g^{\uparrow}$ have been given the name *coinductive predicate*, or *co-predicate* for short. The main technique for proving something from a given proof tree, that is, to prove something of the form $g^{\uparrow}(x) \implies R$, is to destruct the proof. Since this is just unfolding the defining equation, the fact that we're using a greatest solution for g never plays a role (as long as we limit ourselves to a finite number of unfoldings).

To go in the other direction, to establish a predicate defined as a greatest solution, like $Q \implies g^{\uparrow}(x)$, we may need an infinite number of steps. For this purpose, we can use induction's dual, *coinduction*. Were it not for one little detail, coinduction is as simple as continuations in programming: the next part of the proof obligation is delegated to the *coinduction hypothesis*. The little detail is making sure that it is the "next" part we're passing on for the continuation, not the same part. This detail is called *productivity* and corresponds to the requirement in induction of making sure we're going down a well-founded relation when applying the induction hypothesis. There are many sources with more information, see for example the classic account by Jacobs and Rutten [8] or a new attempt by Kozen and Silva that aims to emphasize the simplicity, not the mystery, of coinduction [10].

Let's prove $\mathit{true} \implies g^{\uparrow}(x)$. The intuitive coinductive proof goes like this: According to the right-hand rule of (10), $g^{\uparrow}(x)$ follows if we establish $g^{\uparrow}(x-2)$, and that's easy to do by invoking the coinduction hypothesis. The "little detail", productivity, is satisfied in this proof because we applied a rule in (10) before invoking the coinduction hypothesis.

For anyone who may have felt that the intuitive proof felt too easy, here is a formal proof using (15), which relies only on induction. We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & Q \implies f^{\uparrow}(x) \\
= & \quad \{ \text{ (15) } \} \\
 & Q \implies \forall k \bullet {}^{\sharp}f_k(x) \\
= & \quad \{ \text{ distribute } \implies \text{ over } \forall \text{ to the right } \} \\
 & \forall k \bullet Q \implies {}^{\sharp}f_k(x)
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove $\mathit{true} \implies {}^{\sharp}g_k(x)$ for every k. If $k = 0$, then ${}^{\sharp}g_k(x)$ is *true*, so our goal holds trivially. If $k > 0$, then ${}^{\sharp}g_k(x) = (x = 0 \ \vee \ {}^{\sharp}g_{k-1}(x-2))$. We establish the second disjunct by applying the induction hypothesis (on the smaller $k-1$ and with $x-2$).

1.3. Other Techniques

Although in this paper I consider only well-founded functions and extreme predicates, it is worth mentioning that there are additional ways of making sure that the set of solutions to (0) is nonempty. For example, if all calls to f in $\mathcal{F}'(f)$ are *tail-recursive calls*, then (under the assumption that Y is nonempty) the set of solutions is nonempty. To see this, consider an attempted evaluation of $f(x)$ that fails to determine a definite result value because of an infinite chain of calls that applies f to each value of some subset X' of X. Then, apparently, the value of f for any one of the values in X' is not determined by the equation, but picking any particular result values for these makes for a consistent definition. This was pointed out by Manolios and Moore [18]. Functions can be underspecified in this way in the proof assistants ACL2 [9] and HOL [11].

2. Functions in Dafny

In this section, I explain with examples the support in Dafny[0] for well-founded functions, extreme predicates, and proofs regarding these.

2.0. Well-founded Functions in Dafny

Declarations of well-founded functions are unsurprising. For example, the Fibonacci function is declared as follows:

```
function fib(n: nat): nat
{
  if n < 2 then n else fib(n-2) + fib(n-1)
}
```

Dafny verifies that the body (given as an expression in curly braces) is well defined. This includes decrement checks for recursive (and mutually recursive) calls. Dafny predefines a well-founded relation on each type and extends it to lexicographic tuples of any (fixed) length. For example, the well-founded relation $x \ll y$ for integers is $x < y \ \wedge \ 0 \leq y$, the one for reals is $x \leq y - 1.0 \ \wedge \ 0.0 \leq y$ (this is the same ordering as for integers, if you read the integer relation as $x \leq y - 1 \ \wedge \ 0 \leq y$), the one for inductive datatypes is structural inclusion, and the one for coinductive datatypes is $false$.

Using a `decreases` clause, the programmer can specify the term in this predefined order. When a function definition omits a `decreases` clause, Dafny makes a simple guess. This guess (which can be inspected by hovering over the function name in the Dafny IDE) is very often correct, so users are rarely bothered to provide explicit `decreases` clauses.

If a function returns `bool`, one can drop the result type `: bool` and change the keyword `function` to `predicate`.

2.1. Proofs in Dafny

Dafny has `lemma` declarations. These are really just special cases of methods: they can have pre- and postcondition specifications and their body is a code block. Here is the lemma we stated and proved in Section 1.0.0:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
```

[0] Dafny is open source at dafny.codeplex.com and can also be used online at rise4fun.com/dafny.

```
{
  if n < 2 {
  } else {
    FibProperty(n-2); FibProperty(n-1);
  }
}
```

The postcondition of this lemma (keyword `ensures`) gives the proof goal. As in any program-correctness logic (e.g., [6]), the postcondition must be established on every control path through the lemma's body. For `FibProperty`, I give the proof by an `if` statement, hence introducing a case split. The then branch is empty, because Dafny can prove the postcondition automatically in this case. The else branch performs two recursive calls to the lemma. These are the invocations of the induction hypothesis and they follow the usual program-correctness rules, namely: the precondition must hold at the call site, the call must terminate, and then the caller gets to assume the postcondition upon return. The "proof glue" needed to complete the proof is done automatically by Dafny.

Dafny features an aggregate statement using which it is possible to make (possibly infinitely) many calls at once. For example, the induction hypothesis can be called at once on all values `n'` smaller than `n`:

```
forall n' | 0 <= n' < n {
  FibProperty(n');
}
```

For our purposes, this corresponds to *strong induction*. More generally, the `forall` statement has the form

```
forall k | P(k)
  ensures Q(k)
{ Statements; }
```

Logically, this statement corresponds to *universal introduction*: the body proves that `Q(k)` holds for an arbitrary `k` such that `P(k)`, and the conclusion of the `forall` statement is then $\forall k \bullet \; P(k) \implies Q(k)$. When the body of the `forall` statement is a single call (or `calc` statement), the `ensures` clause is inferred and can be omitted, like in our `FibProperty` example.

Lemma `FibProperty` is simple enough that its whole body can be replaced by the one `forall` statement above. In fact, Dafny goes one step further: it automatically inserts such a `forall` statement at the beginning of every lemma [13]. Thus, `FibProperty` can be declared and proved simply by:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
{ }
```

Going in the other direction from universal introduction is existential elimination, also known as Skolemization. Dafny has a statement for this, too: for any variable `x` and boolean expression `Q`, the *assign such that* statement `x :| Q;` says to assign to `x` a value such that `Q` will hold. A proof obligation when using this statement is to show that there exists an `x` such that `Q` holds. For example, if the fact $\exists k \bullet \; 100 \leq \mathit{fib}(k) < 200$ is known, then the statement `k :| 100 <= fib(k) < 200;` will assign to `k` some value (chosen arbitrarily) for which `fib(k)` falls in the given range.

2.2. Extreme Predicates in Dafny

In this previous subsection, I explained that a `predicate` declaration introduces a well-founded predicate. The declarations for introducing extreme predicates are `inductive predicate` and `copredicate`. Here is the definition of the least and greatest solutions of g from above, let's call them `g` and `G`:

```
inductive predicate g(x: int) { x == 0 || g(x-2) }
copredicate G(x: int) { x == 0 || G(x-2) }
```

When Dafny receives either of these definitions, it automatically declares the corresponding prefix predicates. Instead of the names ${}^{\flat}g_k$ and ${}^{\sharp}g_k$ that I used above, Dafny names the prefix predicates `g#[k]` and `G#[k]`, respectively, that is, the name of the extreme predicate appended with `#`, and the subscript is given as an argument in square brackets. The definition of the prefix predicate derives from the body of the extreme predicate and follows the form in (11) and (12). Using a faux-syntax for illustrative purposes, here are the prefix predicates that Dafny defines automatically from the extreme predicates `g` and `G`:

```
predicate g#[_k: nat](x: int) { _k != 0 && (x == 0 || g#[_k-1](x-2)) }
predicate G#[_k: nat](x: int) { _k != 0 ==> (x == 0 || G#[_k-1](x-2)) }
```

The Dafny verifier is aware of the connection between extreme predicates and their prefix predicates, (14) and (15).

Remember that to be well defined, the defining functor of an extreme predicate must be monotonic, and for (14) and (15) to hold, the functor must be continuous. Dafny enforces the former of these by checking that recursive calls of extreme predicates are in positive positions. The continuity requirement comes down to checking that they are also in *continuous positions*: that recursive calls to inductive predicates are not inside unbounded universal quantifiers and that recursive calls to co-predicates are not inside unbounded existential quantifiers [15, 19].

2.3. Proofs about Extreme Predicates

From what I have presented so far, we can do the formal proofs from Sections 1.2.0 and 1.2.1. Here is the former:

```
lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
{
  var k: nat :| g#[k](x);
  EvenNatAux(k, x);
}
lemma EvenNatAux(k: nat, x: int)
  requires g#[k](x)
  ensures 0 <= x && x % 2 == 0
{
  if x == 0 { } else { EvenNatAux(k-1, x-2); }
}
```

Lemma `EvenNat` states the property we wish to prove. From its precondition (keyword `requires`) and (14), we know there is some `k` that will make the condition in the assign-such-that statement true. Such a value is then assigned to `k` and passed to the auxiliary lemma, which promises to establish the proof goal. Given the condition `g#[k](x)`, the definition of `g#` lets us conclude `k != 0` as well as the disjunction `x == 0 || g#[k-1](x-2)`. The then branch considers the case of the first disjunct, from which the proof

goal follows automatically. The else branch can then assume `g#[k-1](x-2)` and calls the induction hypothesis with those parameters. The proof glue that shows the proof goal for `x` to follow from the proof goal with `x-2` is done automatically.

Because Dafny automatically inserts the statement

```
forall k', x' | 0 <= k' < k && g#[k'](x') {
  EvenNatAux(k', x');
}
```

at the beginning of the body of `EvenNatAux`, the body can be left empty and Dafny completes the proof automatically.

Here is the Dafny program that gives the proof from Section 1.2.1:

```
lemma Always(x: int)
  ensures G(x)
{ forall k: nat { AlwaysAux(k, x); } }
lemma AlwaysAux(k: nat, x: int)
  ensures G#[k](x)
{ }
```

While each of these proofs involves only basic proof rules, the setup feels a bit clumsy, even with the empty body of the auxiliary lemmas. Moreover, the proofs do not reflect the intuitive proofs I described in Section 1.2.0 and 1.2.1. These shortcoming are addressed in the next subsection.

2.4. Nicer Proofs of Extreme Predicates

The proofs we just saw follow standard forms: use Skolemization to convert the inductive predicate into a prefix predicate for some `k` and then do the proof inductively over `k`; respectively, by induction over `k`, prove the prefix predicate for every `k`, then use universal introduction to convert to the coinductive predicate. With the declarations `inductive lemma` and `colemma`, Dafny offers to set up the proofs in these standard forms. What is gained is not just fewer characters in the program text, but also a possible intuitive reading of the proofs. (Okay, to be fair, the reading is intuitive for simpler proofs; complicated proofs may or may not be intuitive.)

Somewhat analogous to the creation of prefix predicates from extreme predicates, Dafny automatically creates a *prefix lemma* `L#` from each "extreme lemma" `L`. The pre- and postconditions of a prefix lemma are copied from those of the extreme lemma, except for the following replacements: For an inductive lemma, Dafny looks in the precondition to find calls (in positive, continuous positions) to inductive predicates `P(x)` and replaces these with `P#[_k](x)`. For a co-lemma, Dafny looks in the postcondition to find calls (in positive, continuous positions) to co-predicates `P` (including equality among coinductive datatypes, which is a built-in co-predicate) and replaces these with `P#[_k](x)`. In each case, these predicates `P` are the lemma's *focal predicates*.

The body of the extreme lemma is moved to the prefix lemma, but with replacing each recursive call `L(x)` with `L#[_k-1](x)` and replacing each occurrence of a call to a focal predicate `P(x)` with `P#[_k-1](x)`. The bodies of the extreme lemmas are then replaced as shown in the previous subsection. By construction, this new body correctly leads to the extreme lemma's postcondition.

Let us see what effect these rewrites have on how one can write proofs. Here are the proofs of our running example:

```
inductive lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
```

```
{ if x == 0 { } else { EvenNat(x-2); } }
colemma Always(x: int)
  ensures G(x)
{ Always(x-2); }
```

Both of these proofs follow the intuitive proofs given in Sections 1.2.0 and 1.2.1. Note that in these simple examples, the user is never bothered with either prefix predicates nor prefix lemmas—the proofs just look like "what you'd expect".

Since Dafny automatically inserts calls to the induction hypothesis at the beginning of each lemma, the bodies of the given extreme lemmas `EvenNat` and `Always` can be empty and Dafny still completes the proofs. Folks, it doesn't get any simpler than that!

3. Case Study: Modeling Semantics

Computer scientists in the programming language area like to model the semantics of languages in order to reason about their behavior. This activity is often aided by the use of inductive predicates [3, 20, 24, 27], and sometimes coinductive predicates [17]. Let me illustrate with a small excerpt from a semantics proof how Dafny's features can be used.

Chapter 7 of Nipkow and Klein's book *Concrete Semantics* [20] defines the big-step and small-step semantics for the rudimentary imperative language IMP [27]. The commands (statements) of the language are defined using an inductive datatype:

```
datatype com = SKIP | Assign(vname, aexp) | Seq(com, com)
             | If(bexp, com, com) | While(bexp, com)
```

and the big-step semantics is defined using an inductive predicate, of which I show the case for sequential composition (`Seq`) here:

```
inductive predicate big_step(c: com, s: state, t: state)
{ match c ...
  case Seq(c0, c1) =>
    ∃ s' :: big_step(c0, s, s') && big_step(c1, s', t)
}
```

This case corresponds to what in inference rules would be rendered as follows:

$$\frac{big\_step(c0, s, s') \qquad big\_step(c1, s', t)}{big\_step(Seq(c0, c1), s, t)} \tag{16}$$

Note how the s' above the line becomes existentially quantified in the definition of the inductive predicate in Dafny.

The small-step semantics of IMP is defined using two inductive predicates, one for a single step (omitted here) and one for the reflexive transitive closure thereof:

```
inductive predicate small_step_star(c: com, s: state, c': com, s': state)
{
  (c == c' && s == s') ||
  ∃ c", s" :: small_step(c, s, c", s") && small_step_star(c", s", c', s')
}
```

A usual theorem of interest is to prove a correspondence between the big-step and small-step semantics. Here, I show the statement of that theorem along with the proof case for `Seq`:

```
inductive lemma BigStep_implies_SmallStepStar(c: com, s: state, t: state)
  requires big_step(c, s, t)
  ensures small_step_star(c, s, SKIP, t)
{ match c ...
  case Seq(c0, c1) =>
    var s' :| big_step(c0, s, s') && big_step(c1, s', t);
    calc {
      true;
    ==>  // induction hypothesis
      small_step_star(c1, s', SKIP, t);
    ==>  // small-step semantics with SKIP as first argument to Seq
      small_step_star(Seq(SKIP, c1), s', SKIP, t);
    ==>  // induction hypothesis
      small_step_star(c0, s, SKIP, s') && small_step_star(Seq(SKIP, c1), s', SKIP, t);
    ==>  { lemma_7_13(c0, s, SKIP, s', c1); }
      small_step_star(Seq(c0, c1), s, Seq(SKIP, c1), s') &&
      small_step_star(Seq(SKIP, c1), s', SKIP, t);
    ==>  { star_transitive(Seq(c0, c1), s, Seq(SKIP, c1), s', SKIP, t); }
      small_step_star(c, s, SKIP, t);
    }
}
```

This proof case first Skolemizes the `s'` from the corresponding definition of `big_step` and then embarks on a proof calculation [16] that shows successive implications from `true` to the proof goal. The proof looks natural and never mentions any prefix predicate explicitly. Under the hood, the lemma's precondition is really `big_step#[_k](c, s, t)` and the right-hand side of the Skolemization is really `big_step#[_k-1](c0, s, s') && big_step#[_k-1](c1, s', t)`. The first and third steps of the calculation hold on behalf of these prefix predicates and the (automatically applied) induction hypothesis. Overall, these shorthands contribute to a short and readable proof.

One more example will illustrate a final point. Here is the statement and proof of "Lemma 7.13" that was used above:

```
inductive lemma lemma_7_13(c0: com, s0: state, c: com, t: state, c1: com)
  requires small_step_star(c0, s0, c, t)
  ensures small_step_star(Seq(c0, c1), s0, Seq(c, c1), t)
{
  if c0 == c && s0 == t {
  } else {
    var c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t);
    lemma_7_13(c', s', c, t, c1);
  }
}
```

The else branch of this lemma, like the `Seq` case in the proof above, uses Skolemization to give names (`c'` and `s'`) to what is known to hold at this point. This is typical when the definition of the inductive predicate uses an existential quantifier. But what if the definition has further disjuncts with existential quantifiers? Then the `if` statement in the lemma must check for these, which I can illustrate with the same example by just reversing the order of the then and else branches:

```
if ∃c', s' :: small_step(c0, s0, c', s') && small_step_star(c', s', c, t) {
  var c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t);
```

```
  lemma_7_13(c', s', c, t, c1);
}
```

(Here, I omitted the else branch in the usual way, since it is empty anyway.) Having to repeat the condition both in the `if` guard and the subsequent Skolemization is clumsy. Therefore, Dafny features `if` statements with binding guards, which allow the body of `lemma_7_13` to be simply:

```
if c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t) {
  lemma_7_13(c', s', c, t, c1);
}
```

In short, the `if` alternative is taken if there exist values for `c'` and `s'` that make the condition hold, and then `c'` and `s'` remain bound in the then branch. Dafny also includes a symmetric `if` statement, like the **if** ... **fi** statement in Dijkstra's guarded command language [4]. It also supports the binding guards, as can be seen here:

```
if {
  case c0 == c && s0 == t =>
  case c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t) =>
    lemma_7_13(c', s', c, t, c1);
}
```

This concludes my tutorial examples. More examples of coinductive definitions and proofs are found in previous papers [14, 15]. The full Dafny encoding of Nipkow and Klein's chapter 7 is found in the test suite of the Dafny open-source distribution, dafny.codeplex.com. Never in this encoding (other than in an example) are the prefix predicates or prefix lemmas mentioned explicitly, which gives support to the idea that the syntactic rewrites hit the spot. A user can inspect the rewrites by hovering over the calls to the extreme lemmas and predicates in the Dafny IDE.

4. Other Tools

Other tools, like Coq [22], Isabelle [21], HOL [5], Agda [1], VeriFast [7], and F* [25], have since long supported inductive predicates, typically via dependent types. In these languages, the notation for defining inductive predicates is inverted compared to Dafny, using a *clausal form* rather than a *casewise form* [5]. For example, here is the big-step definition in Coq, showing the case for `Seq`:

```
Inductive big_step : com -> state -> state -> Prop := ...
  | BS_Seq : forall c0 c1 s s' t,
      big_step c0 s s' -> big_step c1 s' t -> big_step (Seq c0 c1) s t
```

Sometimes, this direction of the definition is more intuitive. It would be nice to support in Dafny an alternative syntax for writing definitions this way.

In this paper, I have presented extreme predicates as being defined as extreme fixpoints of a functor $\mathcal{F}$. A well-founded predicate is also a fixpoint of the defining functor, but talking about it as a least or greatest fixpoint is not interesting, since the fixpoint of the defining functor is unique. From this perspective, it is curious that the keyword used in Coq to define a well-founded function is `Fixpoint`.

Another difference is that whereas tools like Coq and F* do the induction over the actual proof tree, Dafny's induction is essentially over the *height* of the proof tree (an upper bound of which is given by `_k`). With the syntactic rewriting shown in this paper, the Dafny proofs can read as if they were over the proof trees, rather than having to talk about the height explicitly.

When it comes to defining co-predicates, the continuity restriction is awkward. It means that the common existential quantifiers like in the inductive definition of `Seq` above cannot be used directly. The

workaround is to move the existential quantifier outside the entire co-predicate, see [15]. It would be wonderful to have a different solution for this in Dafny.

The fact that the Dafny verifier uses an SMT solver provides useful automation for a lot of proof glue. Dafny's encoding of extreme predicates and its automatic insertion of the induction hypothesis extend this automation to more advanced proof steps. The verifying type checker for F* [25] also uses an SMT solver, but does not include the more advanced automation. The Why3 language provides a syntax for inductive predicates and its verifier backend supports several SMT solvers. However, the inductive predicates defined are treated as any arbitrary solution to (0), so the SMT solvers do not reason about least or greatest fixpoints [3].

In this paper, I've talked about Dafny as a tool to state and prove lemmas. More generally, Dafny is a programming language and the constructs I have shown for proofs are shared with the compiled fragment of the language. In particular, forms of the `forall` statement, the assign-such-that statement, and the binding `if` guards are also available for writing programs that compile and run.

5. Conclusions

In this paper, I have conveyed a way to understand well-founded functions and extreme predicates and have given a number of small but representative examples of their use. The Dafny language previously had well-founded functions, induction, co-predicates, and co-lemmas. New in this paper are the inductive predicates and inductive lemmas, which are simply the duals of the coinductive counterparts. Having both makes for a more balanced understanding of how these are used, and I tried in my presentation not to make the coinductive constructs seem any more mysterious than the inductive counterparts. Because the inductive constructs are used more often in practice, the additional experience with them has led to the further improvements presented in this paper, namely rewriting of focal predicates in extreme lemmas and the binding `if` guards. I hope that the automation facilitated by Dafny, as well as this tutorial itself, will give students and researchers less painful access to mechanized support around formalizations and proofs.

Acknowledgments Some of the crystalized thinking that went into the presentation in this paper came while preparing for a mini-course on Formal Semantics that I taught at Imperial College London in May 2015. I've always been jealous of the nice inductive predicates in Coq, sometimes philosophically pondering the question *Are they datatypes or functions?*, and I am glad to finally support a form of them (as functions) in Dafny. I thank the referees and Jonathan Protzenko for helpful comments on drafts of this paper and am grateful to Daan Leijen for assistance with type setting.

References

[0] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Comp. Sci. Springer, 2004.

[1] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, Aug. 2009.

[2] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, 1992.

[3] M. Clochard, J.-C. Filliâtre, C. Marché, and A. Paskevich. Formalizing semantics with an automatic program verifier. In *VSTTE 2014*, volume 8471 of *LNCS*, pages 37–51. Springer, July 2014.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[5] J. Harrison. Inductive definitions: Automation and application. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *TPHOLs 1995*, volume 971 of *LNCS*, pages 200–213. Springer, 1995.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580,583, Oct. 1969.

[7] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Dept. of Computer Science, Katholieke Universiteit Leuven, 2008.

[8] B. Jacobs and J. Rutten. An introduction to (co)algebra and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, number 52 in Cambridge Tracts in Theoretical Comp. Sci., pages 38–99. Cambridge Univ. Press, 2011.

[9] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[10] D. Kozen and A. Silva. Practical coinduction. Technical Report http://hdl.handle.net/1813/30510, Comp. and Inf. Science, Cornell Univ., 2012.

[11] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.

[12] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[13] K. R. M. Leino. Automating induction with an SMT solver. In *VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, Jan. 2012.

[14] K. R. M. Leino. Automating theorem proving with SMT. In *ITP 2013*, volume 7998 of *LNCS*, pages 2–16. Springer, July 2013.

[15] K. R. M. Leino and M. Moskal. Co-induction simply — automatic co-inductive proofs in a program verifier. In *FM 2014*, volume 8442 of *LNCS*, pages 382–398. Springer, May 2014.

[16] K. R. M. Leino and N. Polikarpova. Verified calculations. In *VSTTE 2013*, volume 8164 of *LNCS*, pages 170–190. Springer, 2014.

[17] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, Feb. 2009.

[18] P. Manolios and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[19] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[20] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014.

[21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[22] C. Paulin-Mohring. Inductive definitions in the system Coq — rules and properties. In *TLCA '93*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.

[23] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *CADE-12*, volume 814 of *LNCS*, pages 148–161. Springer, 1994.

[24] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. http://www.cis.upenn.edu/~bcpierce/sf, version 3.2 edition, Jan. 2015.

[25] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP 2011*, pages 266–278. ACM, Sept. 2011.

[26] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[27] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.